Micro-architectural Analysis of In-memory OLTP

Utku Sirin* utku.sirin@epfl.ch
Pınar Tözün† ptozun@us.ibm.com
Danica Porobic* danica.porobic@epfl.ch
Anastasia Ailamaki*‡ anastasia.ailamaki@epfl.ch

*École Polytechnique Fédérale de Lausanne
†IBM Almaden Research Center
‡RAW Labs SA

ABSTRACT

Micro-architectural behavior of traditional disk-based online transaction processing (OLTP) systems has been investigated extensively over the past couple of decades. Results show that traditional OLTP mostly under-utilize the available micro-architectural resources. In-memory OLTP systems, on the other hand, process all the data in main-memory, and therefore, can omit the buffer pool. In addition, they usually adopt more lightweight concurrency control mechanisms, cache-conscious data structures, and cleaner codebases since they are usually designed from scratch. Hence, we expect significant differences in micro-architectural behavior when running OLTP on platforms optimized for in-memory processing as opposed to disk-based database systems. In particular, we expect that in-memory systems exploit micro architectural features such as instruction and data caches significantly better than disk-based systems.

This paper sheds light on the micro-architectural behavior of in-memory database systems by analyzing and contrasting it to the behavior of disk-based systems when running OLTP workloads. The results show that despite all the design changes, in-memory OLTP exhibits very similar micro-architectural behavior to disk-based OLTP systems: more than half of the execution time goes to memory stalls where L1 instruction misses and the long-latency data misses from the last-level cache are the dominant factors in the overall stall time. Even though aggressive compilation optimizations can almost eliminate instruction misses, the reduction in instruction stalls amplifies the impact of last-level cache data misses. As a result, the number of instructions retired per cycle barely reaches one on machines that are able to retire up to four for both traditional disk-based and new generation in-memory OLTP.

Keywords

OLTP; Workload characterization; In-memory OLTP systems; Micro-architectural analysis

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882916

1. INTRODUCTION

Recent years have witnessed the rise of the in-memory or main-memory optimized OLTP systems [5, 19, 31]. Traditional OLTP engines are disk-based since they are designed in an era where the server hardware had a main-memory size in megabytes. Today, however, a server hardware with 1TB main-memory is a commodity. Therefore, the database management systems (DBMSs) are able to process the data working set of most OLTP applications in memory. This has led various vendors and researchers to design brand new OLTP engines optimized for the case where the hot dataset resides in memory [12, 14, 15, 27].

In-memory OLTP systems have several significant differences compared to disk-based systems. First, since the data working set resides mostly in memory, they omit the buffer pool component, which acts as the virtual memory of a DBMS and is, therefore, essential for the disk-based systems. Then, they tend to adopt more lightweight concurrency control mechanisms to avoid the scalability bottlenecks that arise due to traditional centralized locking. They also opt for cache-conscious indexes instead of the disk-optimized B-trees. Finally, since their codebases are written from scratch, they tend to have lighter storage engines in terms of the instruction footprint.

OLTP benchmarks are famous for their suboptimal micro-architectural behavior. There is a large body of work that characterizes OLTP benchmarks at the micro-architectural level [3, 6, 11, 23, 25, 28, 29]. They all conclude that OLTP exhibits high stall time ($>$ 50% of the execution cycles), and a low instructions-per-cycle (IPC) value ($<$ 1 IPC on machines that can retire up to 4 instructions in a cycle) [6]. The L1 instruction misses that mainly stem from the large instruction footprint of transactions are the main source of the stall time, while the next contributing factor is the long-latency data misses from the last-level cache (LLC) [28].

All the previous workload characterization studies, however, run the OLTP benchmarks on a disk-based OLTP engine. Considering the lighter components, cache-friendly data structures, and cleaner codebase of in-memory systems, one expects them to exhibit better cache locality (especially for the L1-I cache) and less memory stall time. Due to the distinctive design features of the in-memory systems from the disk-based ones, however, it is not straightforward to extrapolate how OLTP benchmarks behave at the micro-architectural level when run on an in-memory engine solely by looking at the results of previous studies.

In this paper, we perform a detailed analysis of the micro-architectural behavior of the in-memory OLTP systems. More

specifically, we compare three in-memory OLTP systems (VoltDB [31], HyPer [9], and the in-memory OLTP engine of a closed-source commercial vendor) to two disk-based OLTP systems (Shore-MT [24] and a disk-based commercial DBMS) in terms of their IPC values, stall cycles, and cache misses while running simple micro-benchmarks as well as the more complex TPC benchmarks (TPC-B and TPC-C) [1]. Our analysis demonstrates the following:

- Despite all the design differences, in-memory OLTP behaves very similarly to the traditional disk-based OLTP at the micro-architectural level. More specifically, it spends more than half of the execution cycles in memory stalls, mostly due to L1-I misses, and exhibits low IPC.
- Even though the main-memory optimized DBMS components reduce the total instruction footprint at the storage manager side for the in-memory OLTP systems, the instruction footprint and code complexity of the rest of the components overshadow the benefits of these optimizations at the micro-architectural level.
- Transaction-specific compilation optimizations eliminate almost all the L1-I misses. However, the reduction in the instruction related stall time amplifies the impact of long-latency data misses when the data does not fit in LLC, and further reduces the IPC value.

The rest of the paper is organized as follows. Section 2 gives an overview of the in-memory OLTP systems and surveys related work on workload characterization and micro-architectural analysis studies. Section 3 describes the experimental methodology. Section 4 and Section 5 present the analysis results with a micro-benchmark and TPC benchmarks, respectively. Section 6 analyzes the effects of transaction compilation, index structures, and data types, whereas Section 7 investigates the impact of multithreading on the micro-architectural behavior. Finally, Section 8 discusses the results and Section 9 concludes.

2. BACKGROUND AND RELATED WORK

In-memory DBMSs have gained a lot of popularity in the last decade. In Section 2.1, we detail the underlying factors for this trend and the main design characteristics of in-memory OLTP systems. Then, in Section 2.2, we go over the recent workload characterization studies that focus on OLTP applications and highlight why they are not representative for in-memory OLTP systems.

2.1 In-memory OLTP

Commodity servers of the last decade follow two fundamental trends: (1) main-memory becoming cheaper and (2) number of cores increasing exponentially. Simply increasing the buffer pool size and the number of worker threads in the system to exploit the large main-memory and all the available cores, respectively, lead to marginal gains. Therefore, these two hardware trends have triggered alternative design opportunities for the new generation DBMSs.

As DRAM prices become cheap enough to buy 1TB main-memory for ∼$30K, today it is possible for most OLTP applications to keep all of their data working set in main-memory while running on a commodity server hardware. This has led to the development of various in-memory or main-memory optimized OLTP systems. These systems either manage all the data in main-memory or make sure that the hot data resides in main-memory. Since they manage to eliminate/minimize the disk I/O for the data page accesses, the overheads associated with managing the buffer pool overweigh its benefits [8]. Therefore, the in-memory OLTP systems omit the buffer pool component even though it is essential for the traditional disk-based DBMSs as it gives the illusion of an infinite main-memory to the system.

On the other hand, in step with Moore's law, the hardware vendors keep providing more and more opportunities for parallelism. Modern servers tend to have multiple multicore processors in the same machine and allow OLTP systems to handle increasing number of transactional requests in parallel. However, the traditional concurrency control mechanisms that ensure isolation among concurrent transactions using a centralized lock manager and two-phase locking are designed at an era where the server hardware were uniprocessors. Therefore, they do not scale on multicores preventing OLTP systems from exploiting the sheer number of cores available to them [22, 33].

In order to achieve better scalability on multicore architectures, in-memory OLTP systems adopt alternative concurrency control mechanisms. These mechanisms can be broadly grouped into two categories based on whether they partition the data or not. The ones that partition the data choose an extreme form of physical partitioning where there is a data partition for each core and a single worker thread for each partition. Systems like VoltDB [27] (or its ancestor H-Store [26]) and HyPer [12] deploy this approach. As a result, they can avoid any form of locking within a partition and need to coordinate worker threads only when a transaction requires data from multiple partitions (i.e., in the case of distributed transactions). On the other hand, the systems that prefer avoiding any kind of data partitioning, like Hekaton [14] or SAP HANA [15], rely on optimistic and multiversion concurrency control [4].

In addition to alternative concurrency control mechanisms, in-memory database systems also deploy cache-conscious index structures. They align the index page sizes to the size of a cache line as opposed to the size of a disk page and/or adopt lock-free index page access mechanisms rather than using traditional page latches [17, 30]. Moreover, the in-memory OLTP systems tend to depend on pre-determined stored procedures instead of ad-hoc queries [12, 14, 27] and apply efficient compilation optimization techniques that optimize the instruction stream for a particular transaction [14, 21]. Finally, the new-age in-memory OLTP systems have codebases that are implemented from scratch. Therefore, they are expected to have a cleaner codebase compared to the traditional disk-based systems where the codebase consists of many branch statements and patches due to different release versions spanning several decades of development.

Overall, in-memory OLTP engines deploy lighter storage manager components compared to the traditional disk-based systems aiming to utilize the resources of the modern server hardware in a more effective way.

2.2 OLTP at the Micro-architectural Level

There is a large body of related work analyzing the micro-architectural behavior of OLTP workloads. Barrosso et al. [3] investigate the memory system behavior of OLTP and DSS style workloads both on a real machine and with a full-system simulation. They argue that these two types of workloads would benefit from different architectural designs

in terms of the memory system. Ranganathan et al. [23] perform a similar analysis. However, they only focus on the effectiveness of out-of-order execution on SMPs while running these workloads in a simulation environment. On the other hand, Keeton et al. [11] and Stets et al. [25] experiment only with OLTP benchmarks (TPC-B and TPC-C) on real hardware. All of these studies agree that OLTP workloads utilize the underlying micro-architectural resources very poorly, wasting most of the execution cycles on memory stalls and exhibiting a low IPC value.

Ailamaki et al. [2] examine where the time goes on four commercial DBMSs using a micro-benchmark to have a fine-grain understanding of the memory system behavior on multiprocessors, whereas Hardavellas et al. [7] analyze TPC-C and TPC-H on both in-order and out-of-order machines in a simulation environment. These studies focus on the implications for the DBMSs rather than the hardware to achieve better hardware utilization.

More recent workload characterization studies [6, 29] additionally analyze the TPC-E benchmark and show that micro-architecturally TPC-E behaves very similarly to the TPC-B and TPC-C benchmarks. These studies also corroborate the findings of the previous studies in terms of the inefficient use of the memory hierarchy when running OLTP. They highlight that the L1-I stalls are the dominant factor in the overall stall time followed by the long-latency data misses. Our experimental methodology while measuring various hardware events using counters on real hardware is very similar to the methodologies of these studies.

Harizopoulos et al. [8] demonstrate that traditional OLTP systems spend more than half of their execution time within the buffer pool, latching, locking, and logging components. On the other hand, Wenisch et al. [32] and Tozun et. al [28] tie the micro-architectural behavior of the disk-based OLTP into specific code modules by presenting the breakdown of the cache misses into specific code parts of the traditional OLTP software stack at different code granularities.

As Section 2.1 explains, the in-memory OLTP systems either remove or simplify most of the traditional disk-based OLTP components. Therefore, the micro-architectural behavior of OLTP workloads when run on disk-based systems cannot be representative for the in-memory systems. Even worse, the previous findings might mislead researchers and developers that aim to improve utilization at the micro-architectural level when running OLTP workloads using in-memory OLTP systems. Therefore, the focus of this paper is to perform a workload characterization study for OLTP benchmarks running on in-memory OLTP systems to understand the low-level differences between in-memory and disk-based OLTP; and based on these findings, provide valuable insights for OLTP systems' design.

3. SETUP AND METHODOLOGY

The experiments presented in this paper are executed on real hardware and performance is measured using event counters as opposed to hardware simulators since we are not investigating the impact of changing some of the hardware parameters on the micro-architectural behavior. The rest of this section details the setup and methodology for our study.

Hardware: We run experiments on a modern commodity server with Intel's Ivy Bridge processors. Table 1 shows the architectural details of this server. To collect numbers about various hardware events and break down the time spent in specific code modules, we use Intel VTune Amplifier XE 2015 [10], which provides an API for lightweight hardware counter sampling. We disable hyper-threading to obtain more precise hardware sampling values and increase predictability in measurements.

Table 1: Server Parameters

Processor	Intel(R) Xeon(R) CPU E5-2640 v2 (Ivy Bridge)
#Sockets	2
#Cores per Socket	8
#HW Contexts	16
Hyper-threading	Off
Clock Speed	2.00GHz
Memory	256GB
L1I / L1D (per core)	32KB / 32KB 8-cycle miss latency
L2 (per core)	256KB 19-cycle miss latency
LLC (shared)	20MB 167-cycle miss latency

OS: We run all the experiments using RHEL 6.5 with Linux kernel version 2.6.32.

Benchmarks: We run two types of benchmarks: micro-benchmarks and TPC benchmarks [1]. Our goal is to perform sensitivity analysis and have a more detailed understanding of the systems using the micro-benchmark, while the experiments using the TPC benchmarks serve to give an idea about the behavior of the systems when running well-known real-world applications.

The micro-benchmark uses a randomly generated table with two columns (`key` and `value`) of the type *Long*. It has two versions: read-only and read-write. The read-only version reads N random rows from the table, whereas the read-write version updates N random rows. Both versions use an index lookup operation on the randomly picked `key` value to reach the row to be read or updated. We also use a modified version of the micro-benchmark where we use the type *String* for both columns to quantify the impact of data type on micro-architectural utilization in Section 6.2.

As for the TPC benchmarks, we use TPC-B and TPC-C. We omit the more recent TPC-E benchmark since recent workload characterization studies demonstrate that TPC-E exhibits similar micro-architectural behavior to the TPC-B and TPC-C benchmarks [6, 29].

Analyzed Systems: We analyze three in-memory OLTP systems: *VoltDB* [31] (Community Edition Version 4.8), *HyPer* [9] (online demo-version), and the in-memory OLTP engine of a closed-source commercial vendor (*DBMS M*).

We pick these three systems as they are well-known in the community and their design characteristics represent a good variety among today's in-memory OLTP systems. While VoltDB and HyPer use physical data partitioning, DBMS M adopts optimistic multiversioned concurrency control. VoltDB uses traditional B-tree with node size tuned to the last-level cache line size [26]. HyPer implements adaptive radix tree with adaptive compact node sizes [16]. DBMS M implements both hash index and a variant of cache-conscious B-tree index similar to [17, 18]. For this system, we use the hash index for micro-benchmarks and TPC-B, and the B-tree index for TPC-C. Lastly, HyPer and DBMS M use

transaction compilation techniques for the stored procedures [21], whereas VoltDB does not.

In order to gain better insights about the differences between the in-memory and disk-based OLTP systems, we also include two disk-based systems: the open-source *Shore-MT* [24] storage manager and a commercial system (*DBMS D*).

To implement benchmarks, we use VoltDB's Java-based developer front-end, HyPer's SQL-based programming language HyPerScript, Shore-MT's Shore-Kits suite that provides an environment to implement benchmarks for Shore-MT in C++, and the SQL frontend the closed-source commercial systems, DBMS M and DBMS D.

For all the systems, we use asynchronous logging. Therefore, there is no delay due to I/O in the critical path of the transaction execution.

Measurements: We populate the databases from scratch before each experiment and the data remains memory-resident throughout the experiment. In the following sections, we indicate the database sizes used in each experiment before discussing the results. In our experiments, both the database server process executing the transactions and the client processes generating the transactions run on the same machine. We first start the server process, populate the database, and then start the experiment by simultaneously launching all clients that generate and submit transactional requests to the database server.

We profile the database server process by attaching VTune to it during a 60-second benchmark run following a 60-second warm-up period. We report VTune counter results that correspond to the middle 30 seconds of the 60-second run in order to eliminate any peripheral effect that might happen during the initial or final part of the run. We repeat every experiment three times and report the average result.

In terms of micro-architectural efficiency, our goal is to observe how well each system exploits the resources of a single core regardless of the parallelism in the system. Therefore, all the experiments except for the ones in Section 7, use a single worker thread executing the transactions of the corresponding benchmark.

The choice of a single worker thread also eliminates contention due to several threads trying to access the shared data in the case of non-partitioned systems and distributed transactions in the case of partitioning-based systems. This way we avoid possible misleading micro-architectural conclusions. For example, high contention for a shared data page could lead to multiple threads spinning on a latch for that data page, thus artificially increasing the cache hit ratio.

We use one client to generate request in the single-threaded experiments. *Shore-MT*, *DBMS D*, *HyPer*, and *DBMS M* assign one worker thread per client. *VoltDB*, on the other hand, generates one worker thread per data partition, so we configure it to have only one partition. From VTune, we filter the hardware counter results particularly for the identified worker thread excluding the other threads that are responsible for background tasks, e.g., communication between the server and client, parsing transactions, etc.

In multi-threaded experiments (Section 7), we use multiple clients to generate requests for all systems except for *HyPer* whose online demo-version only supports single-client and single-threaded execution. For *VoltDB*, we also use multiple data partitions and ensure that all transactions access only a single partition. For each system, we gradually increase the number of clients, and profile the execution

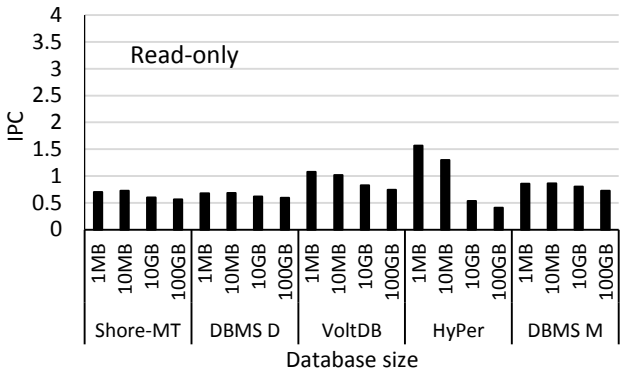

Figure 1: Effect of database size on the IPC value.

with the number of clients that give the highest aggregate throughput. From VTune, we filter hardware counter results for each worker thread separately and report their average.

Using the hardware performance counters, we measure the number instructions retired per cycle (IPC), and the instruction and data stall cycles from all the levels of the memory hierarchy. While reporting the stall cycles, we multiply the number of misses from each cache level with the expected penalty for that particular miss as specified in Table 1. For the last-level cache (LLC) misses, we average the penalty of going to local and remote memory. We note that one cannot be precise while showing the stall cycles breakdown on an out-of-order processor due to the overlapping of different execution components. Therefore, we draw the stall cycles due to different misses side-by-side rather than on top of each other.

4. MICRO-BENCHMARK

Before performing an analysis using the community standard TPC benchmarks, we devise a sensitivity study on the micro-architectural behavior of in-memory OLTP systems using the micro-benchmark. The goal of this study is to answer the following questions:

- How much instruction-level parallelism do OLTP benchmarks exhibit when run on an in-memory system? In other words, what is the number of instructions they can retire per cycle?
- Where do CPU cycles go when running in-memory OLTP? Are they wasted on memory stalls or used to retire instructions?
- Where do memory stalls come from within the memory hierarchy? Are they mainly due to instructions or data for in-memory OLTP?
- What is the impact of the database size on the above metrics?
- Does the amount of work done per transaction affect the results and, if yes, how?

To answer these questions, we break the analysis into two parts. The first part (Section 4.1) varies the database size by varying the number of rows in the table while keeping the amount of work done per transaction constant. On the other hand, the second part (Section 4.2) varies the amount

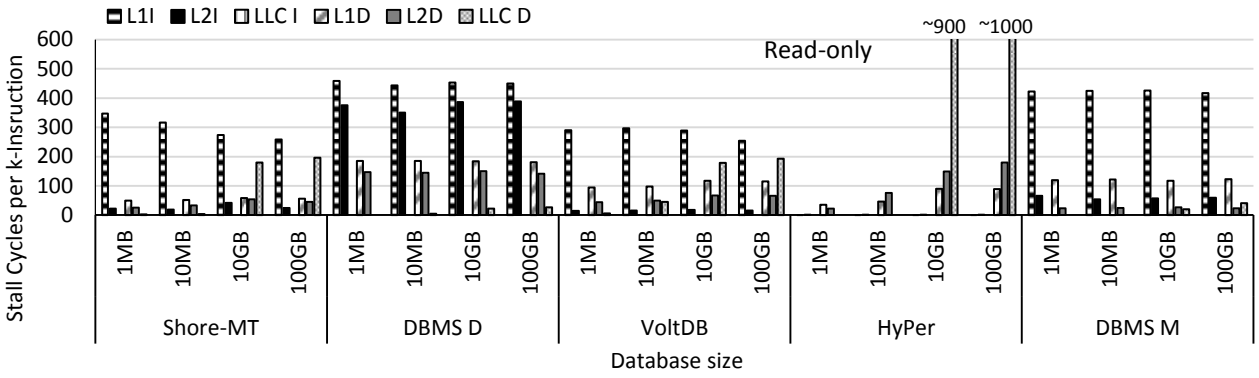

Figure 2: Stall cycles per 1000 instructions from the different levels of the memory hierarchy as we increase the database size.

of work done per transaction by increasing the number of rows read in a transaction while keeping the database size constant. Both parts measure the IPC value and stall cycles that arise due to different types of cache misses.

Since the trends are similar across the read-only and read-write versions of the micro-benchmark, we present the discussions about the read-only version of the micro-benchmark, and place the results and discussions about the read-write version in the Appendix, Section A.

4.1 Sensitivity to Data Size

To investigate the impact of database size on the micro-architectural behavior, we populate databases of size 1MB, 10MB, 10GB, and 100GB. Then, we collect hardware events as the systems run the micro-benchmark with a single transaction type that just reads/updates one random row after an index probe operation. While the results for the read-only version of the micro-benchmark are in the following sub-sections, the results for the read-write version of the micro-benchmark are in Section A.1 of the Appendix.

4.1.1 Instructions-per-Cycle

Figure 1 shows the number of instructions retired per cycle (IPC) on the y-axis for each system as the database size increases on the x-axis. The IPC values are similar for databases of sizes 1MB and 10MB since the data working set mainly fits in the last-level cache, which is 20MB (see Table 1). As we increase the database size to 10GB and 100GB, the IPC decreases since the data working set no longer fits in caches and the long-latency data misses become more significant (as shown in Section 4.1.2).

Where the two disk-based systems (*Shore-MT* and *DBMS D*) exhibit similar behavior in terms of their IPC values, the in-memory systems exhibit a slightly higher IPC value compared to the disk-based systems, except for *HyPer* with large database sizes.

Among the in-memory systems, *VoltDB* achieves a higher IPC value than *DBMS M*. On the other hand, *HyPer* achieves twice as high IPC value compared to the other systems when the data fits in the last-level cache. However, when the data does not fit in the last-level cache, *HyPer* suffers from long-latency data stalls, and hence, it has the lowest IPC value among all the systems.

Overall, even though the server used in our experiments is a 4-way issue one where each core has the ability to retire up to four instructions in a cycle, except for *HyPer* on datasets that fit in last-level cache, both the disk-based and in-memory OLTP systems barely achieve an IPC value of 1.

Since we investigate the impact of various memory-related stalls on the micro-architectural inefficiencies, we also conducted an experiment to measure IPC when there are no instruction or data misses in a program, which would be the ideal scenario in our study. For this experiment, we run a while loop accessing the same two integer variables repeatedly and assigning one's value to the other. The IPC value for this program after its cold start (after the compulsory data and instruction misses are over) is 3. Further investigation of this result is out of the scope of this paper.

4.1.2 Stall Cycles per 1000 instructions

Next, we analyze the causes of the low IPC values. Figure 2 plots the stall cycles per 1000 (k-)instructions coming from all the three levels of the cache hierarchy as we increase the data size on the x-axis. It also separates the instruction and data related stall cycles. For example, *L2D* represents the stall cycles due to data misses from the L2 cache.

From Figure 2, we observe that regardless of the database size, instruction stalls (mainly from *L1I* cache) are the most significant component in the overall stall cycles both for the disk-based and in-memory systems, except for *HyPer*. This shows that, despite all the optimizations described in Section 2.1, in-memory OLTP systems still dramatically suffer from instruction misses similar to the disk-based OLTP systems. On the other hand, these optimizations still help in reducing the instruction related stall time to some extent.

Among the traditional disk-based systems, *Shore-MT*'s instruction stalls are significantly lower than that of *DBMS D*. This is because *Shore-MT* is a storage manager and does not include the layers outside the storage manager component of an OLTP system such as query parser, query optimizer, and communication facilities. It hard-codes the query plan of the transaction in C++.

As the data size increases, *HyPer*'s data stalls increase dramatically, 5-10x more compared to the other systems. This mainly stems from *HyPer*'s high instruction locality. *HyPer* compiles transactions directly into machine code [20, 21]. Therefore, its transactions have an aggressively optimized instruction stream – small instruction footprint, few number of branches in the code, etc. As a result, *HyPer* is able to finish more transactions using the same number

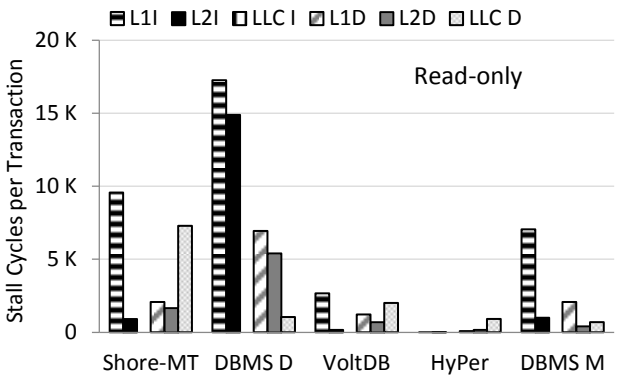

Figure 3: Stall cycles per transaction from the different levels of the memory hierarchy for a database of size 100GB.

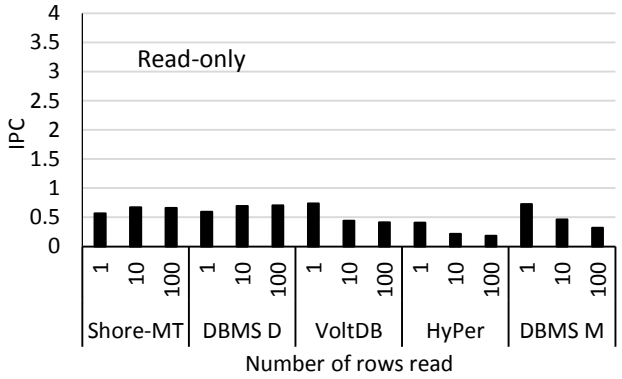

Figure 4: Effect of the amount of work per transaction on the IPC value with a database of size 100GB.

of instructions compared to the other systems. Hence, it touches more data randomly in a unit of time, which causes higher stall time due to long-latency data misses when the whole data set does not fit in *LLC*.

4.1.3 Stall Cycles per Transaction

The number of stall cycles per 1000 instructions provides an aggregated and normalized view of the stall cycles across different systems. In this section, we analyze the stall cycles per single transaction. Figure 3 shows the results for a database of size 100GB only since the results with different database sizes show similar trends.

The first observation is the dramatic decrease in *HyPer*'s LLC data stalls. While the *LLC* data stalls per 1000 instructions are 5-10x higher for *HyPer* than all the other systems, *HyPer*'s data stalls per transaction are among the lowest ones. This shows that *HyPer* executes large number of transactions in a period of time, and makes large number of random data accesses, hence, incurs many data stalls per 1000 instructions.

Except *Shore-MT*, all the other four systems have low *LLC* data stalls per transaction implying that they implement some variant of cache-conscious index structures. *VoltDB* implements a tree index where the node size is tuned to size of the last-level cache line [26]. While *HyPer* uses adaptive radix tree with adaptive compact node sizes [16], *DBMS M* implements hash index. *DBMS D* uses a traditional B-tree with page size of 8KB; however, we could not find any publicly available information about tuning the node size of *DBMS D* for being more cache-conscious. On the other hand, Shore-MT exhibits high *LLC* data stalls due to its non-cache-conscious index structure.

In terms of the instruction stalls, we observe that disk-based *DBMS D*'s instruction stalls are significantly higher than that of the in-memory systems. This highlights that the optimizations in-memory systems adopt help lower the instruction stalls. On the other hand, *DBMS M* has significantly higher *L1I* stalls than *VoltDB* and *HyPer*. This shows the relatively larger instruction footprint of *DBMS M* than that of the other in-memory systems. *DBMS M* is main-memory optimized OLTP engine of a traditional disk-based OLTP system similar to [5, 13, 18]. Therefore, it uses a lot of legacy code for the components outside the storage manager resulting in its relatively higher instruction footprint among the in-memory OLTP systems.

4.2 Sensitivity to Work per Transaction

To investigate the impact of the amount of work per transaction on the micro-architectural behavior, we increase the number of rows that a transaction accesses from 1 to 10 and then to 100. We perform these experiments with 100GB dataset. In the following sub-sections, we present the results for the read-only version of the micro-benchmark. The results for the read-write version of the micro-benchmark can be found in Section A.2 of the Appendix.

4.2.1 Instructions-per-Cycle

Figure 4 presents the IPC values as the number of rows read in a transaction increases on the x-axis. We observe that as the work done per transaction increases, the IPC values of the disk-based systems (*Shore-MT* and *DBMS D*) slightly increase, whereas the IPC values of the in-memory systems (*HyPer*, *VoltDB* and *DBMS M*) decrease. This stems from the changes in the instruction and the data stalls as more rows are read per transaction.

Instruction stalls decrease as the number of rows read increases. This is due to the repetitive work in a transaction which improves the instruction locality. On the other hand, increasing work per transaction in the micro-benchmark leads to more distinct data accesses in a period of time, which hinders data locality in caches and either limit the increase or cause a decrease in the IPC value.

4.2.2 Stall Cycles per 1000 instructions

To better understand the trends in the IPC values in Figure 4, we quantify the stall cycles per 1000 (k-)instructions coming from the different levels of the cache hierarchy similarly to Section 4.1.2. Figure 5 plots the results. The x-axis shows the number of rows read in a transaction.

As briefly explained in Section 4.2.1, the instruction stalls decrease as the number of rows read per transaction increase for all the systems. On the one hand, the repetitive behavior within a transaction leads to a better instruction cache locality. On the other hand, the code for the other layers of the system that surround a transaction's execution (e.g., the code outside the storage manager) is executed less frequently since the transactions get longer as we increase the amount of work done per transaction. For example, where

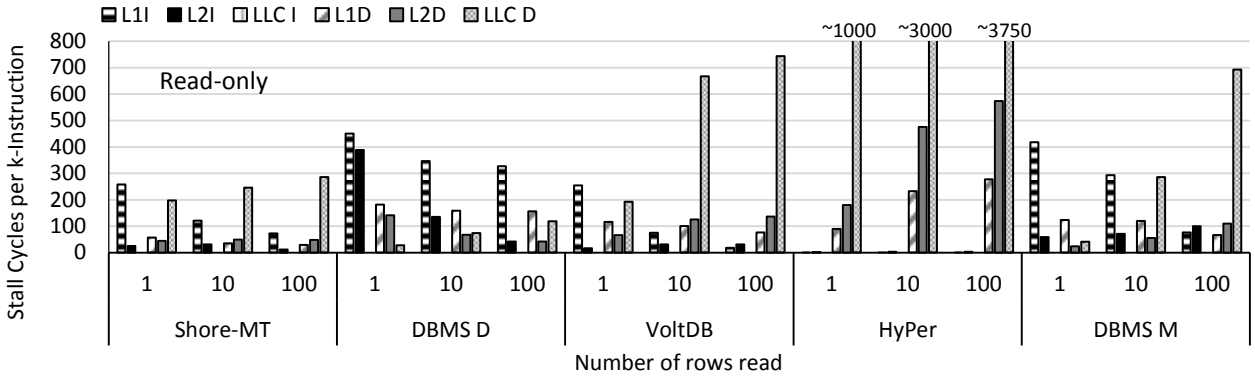

Figure 5: Stall cycles per 1000 instructions from the different levels of the memory hierarchy as we increase the amount of work done per transaction with a database of size 100GB.

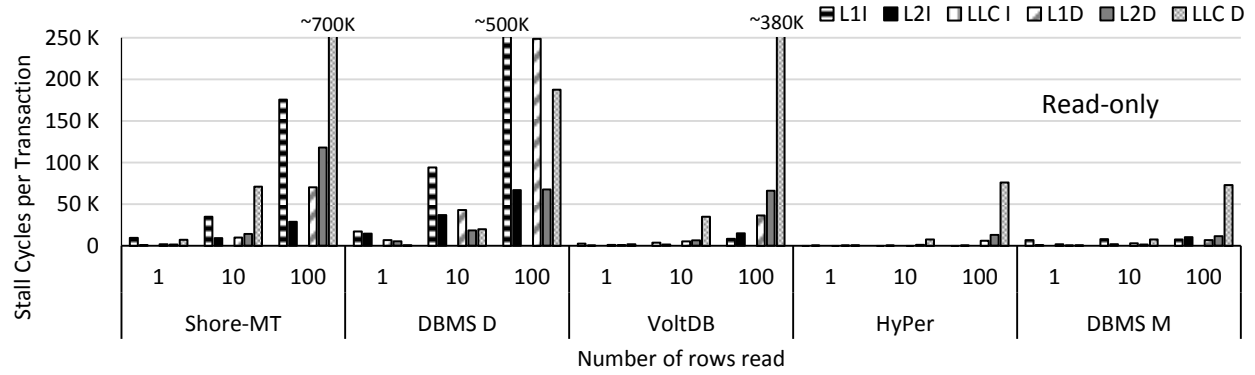

Figure 6: Stall cycles per transaction from the different levels of the memory hierarchy as we increase the amount of work done per transaction with a database of size 100GB.

probing 100 rows per transaction stresses purely the storage manager component of a system, probing 1 row also stresses the other layers such as query parsing, work done while starting/ending a transaction, etc.

While *DBMS M*'s instruction stalls are small for probing 100 rows, they are significantly higher for probing 1 and 10 rows compared to the other in-memory systems. This is because *DBMS M* inherits a lot of legacy code from the traditional disk-based OLTP system it belongs to. This brings a significant overhead on the total instruction footprint that can only be compensated by probing 100 rows per transaction. On the other hand, the low instruction stalls while probing 100 rows shows that the instruction footprint of the *DBMS M*'s storage manager (OLTP engine) is small. One reason for that is *DBMS M* compiles transactions into optimized machine code similar to, but less aggressively than, *HyPer*. Therefore, when stressed at the storage manager side, the compilation optimizations of *DBMS M* are also effective in reducing the instruction stalls. Section 6 quantifies the effect of compilation optimizations in *DBMS M* in more detail.

The data stalls increase as we increase the work done per transaction for all the systems. As we read more random rows per transaction, we access more distinct rows in a period of time. The more frequent random data accesses lead to a higher data miss rate for all the systems. *HyPer* has the highest data stalls for all the cases in Figure 5, since it executes more transactions, and hence, performs more frequent random data accesses compared to the other systems. For *Shore-MT*, *VoltDB*, and *DBMS M*, the *LLC* data stalls are also the dominant contributor to the stall time when accessing many rows per transaction. On the other hand, since *DBMS D* exhibits very high instruction stalls even when probing 100 rows per transaction, its throughput is lower and random data accesses are less frequent. Therefore, *DBMS D*'s LLC data stalls are the lowest.

4.2.3 Stall Cycles per Transaction

To better quantify the impact of accessing more data per unit of time, this section analyzes the stall cycles per transaction as the number of rows read increases from 1 to 100. Figure 6 plots the results. The x-axis shows the number of rows read in a transaction.

Unlike the results in the previous section, instruction stalls per transaction increase as the number of rows accessed in a transaction increases. When the instruction footprint of the loop that probes a random row in each iteration does not fit in the *L1I* and *L2I* caches, increasing the number of rows probed in a transaction increases the number of instruction stalls per transaction. We observe this effect for all the systems except for *HyPer*.

As we increase the number of rows read from 1 to 10, and then to 100, LLC data stalls increase almost linearly for all the systems. We observe that *Shore-MT* has the largest

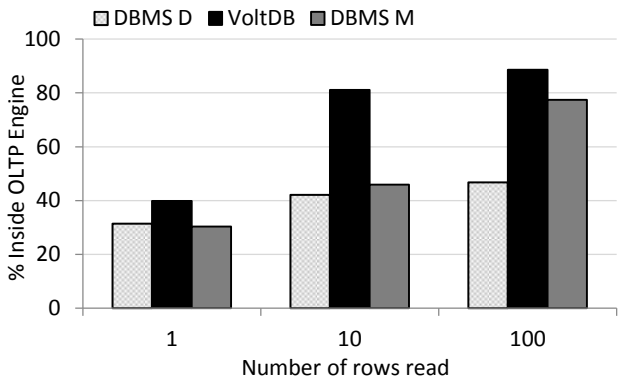

Figure 7: The percentage of the time spent inside the OLTP engine as we increase the amount of work done per transaction with a database of size 100GB.

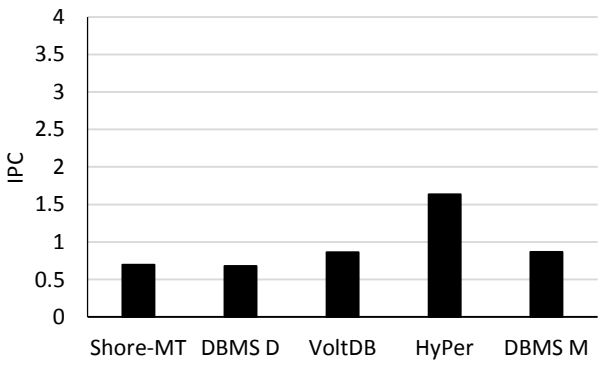

Figure 8: The IPC values while running TPC-B.

data stalls per transaction while probing 100 rows, followed by *VoltDB* and *DBMS D*. *DBMS M*'s hash-based index and *HyPer*'s adaptive radix tree lead to fewer random memory accesses during an index probe compared to the other system's B-tree-based index structures. Therefore, *DBMS M* and *HyPer* have the lowest LLC data stalls per transaction.

4.2.4 Code Modules Breakdown

To better understand the impact of legacy code, as well as components outside the storage manager, we quantify the percentage of the execution time spent in the OLTP engine as the amount of work per transaction increases for the disk-based system *DBMS D*, and the in-memory systems *VoltDB* and *DBMS M*. While performing this breakdown, we have done a best-effort categorization based on the code modules reported by VTune as part of the worker thread execution of each system. Figure 7 shows the results.

We observe that the amount of time spent inside the OLTP engine increases as the number of rows read increases for all three systems. This increase is modest for the disk-based system *DBMS D* showing the high overhead of the code outside the OLTP engine. For *DBMS M*, when we increase the number of rows from 1 to 10, the percentage inside the OLTP engine increases modestly showing the dominance of the legacy code overhead that *DBMS M* borrows from the traditional disk-based OLTP system it belongs to. However, when we increase the number of rows read from 10 to 100, we observe almost 2x increase in the percentage of time spent in the OLTP engine. For *VoltDB*, on the other hand, the amount of time spent inside the storage manager is small for probing 1 row showing the high overhead of the code outside the storage manager for light transactions. However, when we increase the number of probed rows to 10 or 100, the percentage increases more than 2x showing that *VoltDB* compensates this overhead for medium/large-sized transactions.

4.3 Summary

Overall, while the traditional disk-based systems, *Shore-MT* and *DBMS D*, have large instruction footprints and bad instruction locality; the in-memory-optimized systems, *HyPer*, *VoltDB*, and *DBMS M* (when stressed enough on the storage manager) have small, optimized instruction footprints and good instruction locality. The reduction in instruction footprint and the increase in instruction locality, however, lead to more random data accesses in a unit of time and, hence, increase long-latency data stalls when the data working set does not fit in the last-level cache. Therefore, all the systems suffer either from instruction or data stalls resulting in severe under-utilization of the micro-architectural resources having executed barely one instruction-per-cycle on a machine that has the ability of retiring up to four instruction-per-cycle.

5. TPC BENCHMARKS

Section 4 performs a sensitivity analysis using a simple micro-benchmark to gain a fine-grained understanding of the in-memory OLTP systems compared to the disk-based ones at the micro-architectural level. This section investigates the behavior of the same systems while running the more complex and community standard TPC-B (Section 5.1) and TPC-C (Section 5.2) benchmarks. All the experiments in this section use a database of size 100GB. Similar to Section 4, we analyze the IPC values, and the stall cycles per 1000 instructions and per transaction from different levels of the cache hierarchy. Our goal is to observe how the benchmark complexity affects the micro-architectural behavior of the in-memory OLTP systems.

5.1 TPC-B

TPC-B is an update-heavy benchmark that simulates a banking system. `AccountUpdate` is its only transaction type, which updates one row each in three tables, `Branch`, `Teller`, and `Account`, and appends a row to the `History` table.

5.1.1 Instructions-per-Cycle

Figure 8 shows the number of instructions retired per cycle (IPC) on the y-axis as each system runs TPC-B. *HyPer* exhibits the highest IPC value among all the systems, which is in contrast with its IPC being the lowest when running the micro-benchmark (Section 4.2.1). Moreover, we see that the IPC values are in general higher than the IPC values for the read-only micro-benchmark with 100GB data for probing 1 row (Figure 4). This is due to the higher data locality in TPC-B (see also Section 5.1.2). Since `Branch` and `Teller` tables have low cardinality, the probability of a randomly read row from one of these two tables being in *LLC* is high. In addition, `History` table accesses are only appends, which also increases the probability of the `History` table pages be-

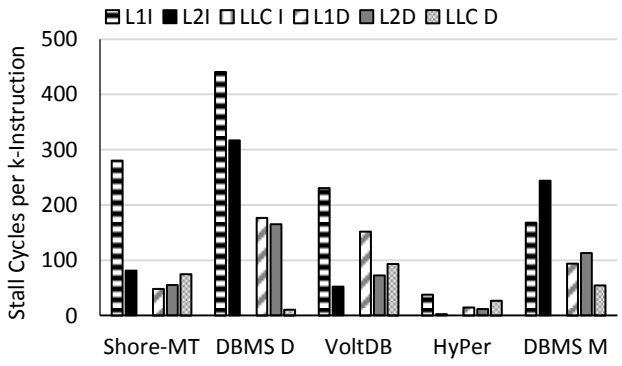

Figure 9: Stall cycles per 1000 instructions while running TPC-B.

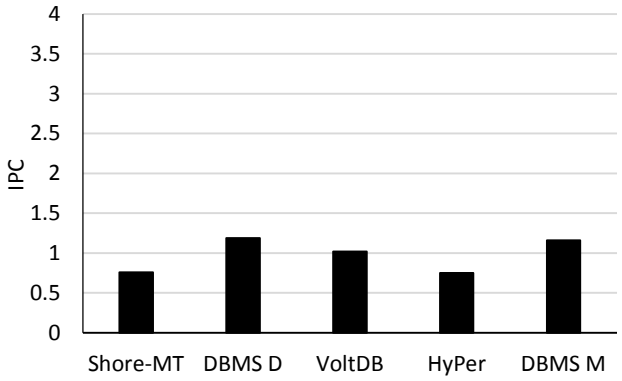

Figure 10: The IPC values while running TPC-C.

ing accessed residing in the caches. Therefore, the only table that exhibits low data locality in TPC-B is the `Account` table.

5.1.2 Stall Cycles

Per 1000 instructions. To investigate the major causes of the stall cycles when running TPC-B, Figure 9 plots the instruction and data stalls per 1000 (k-)instructions for each level of the cache hierarchy for all the systems.

Figure 9 follows similar trends with Figure 5 for instruction stalls for probing 1 row. It demonstrates that the instruction stalls (from both *L1* and *L2* caches) are the dominant factor in the overall stall cycles for all the systems. *DBMS D* has the highest instruction stalls whereas *HyPer* exhibits very high instruction cache locality because of its aggressive query compilation techniques.

On the other hand, none of the systems suffer severely from the long-latency data misses even though we run TPC-B with 100GB data. This is mainly because TPC-B has better data locality compared to the micro-benchmark. When running the micro-benchmark, we randomly probe rows from a 100GB table, which includes more than one billion rows. On the other hand, TPC-B first probes one of the $\sim$ 20K `Branch`es randomly. Then, it probes one of the $\sim$ 200K `Teller`s and one of the $\sim$ 2 billion `Account`s. Finally, it inserts on row into the `History` table. Hence, the probability of re-accessing the same branch or teller as well as the same `History` table page is quite high compared to re-accessing a row from the micro-benchmark's single large table.

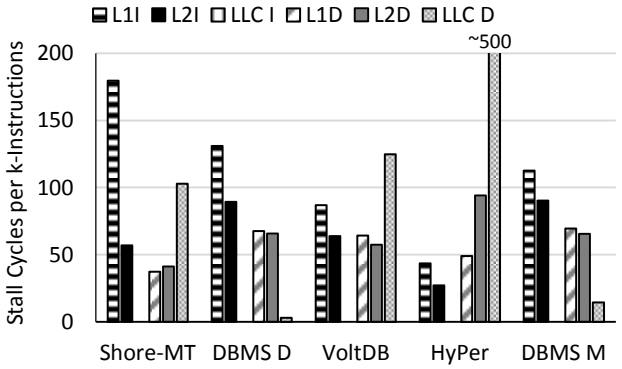

Figure 11: Stall cycles per 1000 instructions while running TPC-C.

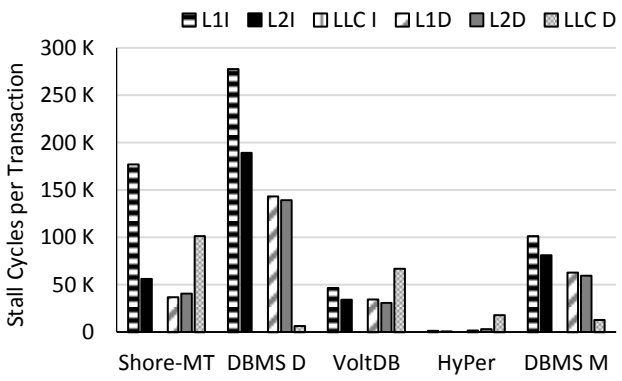

Figure 12: Stall cycles per transaction while running TPC-C.

Per transaction. The stall cycles per transaction follows similar trends with the stall cycles per 1000 instructions. Therefore, we omit the discussion for stall cycles per transaction for TPC-B.

5.2 TPC-C

After investigating the micro-architectural behavior of the systems using TPC-B, this section focuses on the more complex TPC-C benchmark. TPC-C models a wholesale supplier with nine tables and five transaction types (2 of which are read-only and form 8% of the benchmark mix). In terms of the database operations, the TPC-C transactions contain probes, inserts, updates, and joins covering a richer set of operations than TPC-B. Therefore, we expect a different behavior for TPC-C than TPC-B.

5.2.1 Instructions-per-Cycle

Figure 10 reports the IPC values when the systems under analysis run TPC-C. Similar to the TPC-B results, the IPC values are in general higher than the IPC values observed with the micro-benchmark when probing 1 row (Figure 4). Unlike the TPC-B results, however, *DBMS D* and *DBMS M* exhibit the highest IPC among all the systems. In addition, except for *HyPer*, all the systems exhibit a higher IPC value for TPC-C than TPC-B. The differences between the TPC-B and TPC-C results stem from the lower data and higher instruction locality of TPC-C compared to TPC-B which also corroborates the results from [28, 29].

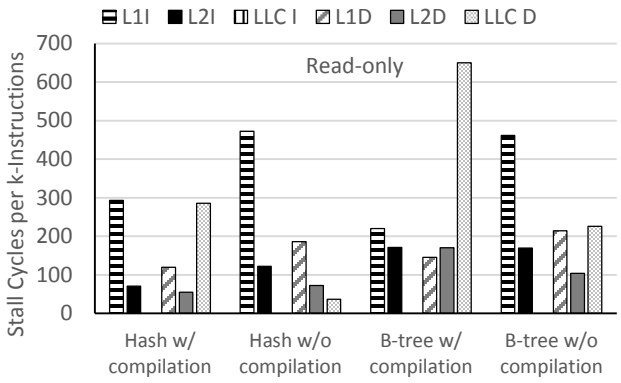

Figure 13: Stall cycles per 1000 instructions for different index structures with and without compilation optimizations while running the micro-benchmark.

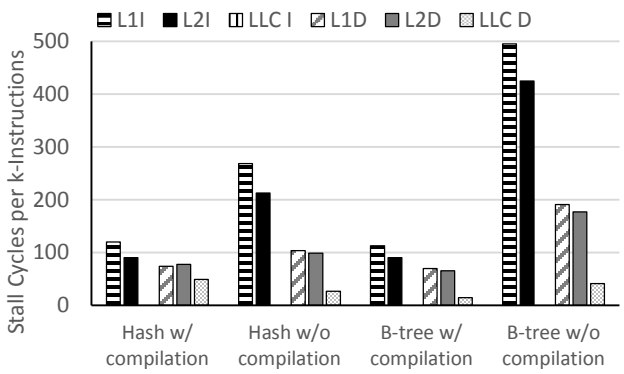

Figure 14: Stall cycles per 1000 instructions for different index structures with and without compilation while running TPC-C.

5.2.2 Stall Cycles

Per 1000 instructions. To better understand the IPC values in Figure 10, Figure 11 shows the instruction and data stalls per 1000 (k-)instructions for each level of the cache hierarchy.

The instruction stall cycles are considerably lower for TPC-C than TPC-B for all the systems. There are two main reasons for this outcome: the longer transactions of TPC-C compared to TPC-B and the number of scan-based accesses in TPC-C. TPC-C's longer transactions reduce the execution frequency of the components outside the storage manager increasing the instruction locality, which is similar to the difference between probing 100 rows vs probing 1 or 10 rows in Section 4.2. TPC-C also performs index-scans in several of its transactions, which increases instruction and data locality for this benchmark since a scan operation is a short loop that keeps fetching the next data item from a page or nearby pages.

With respect to the data stalls, unlike its TPC-B results, *HyPer* exhibits quite high *LLC* data stalls similar to the micro-benchmark experiments, which is due to the lower data locality of TPC-C compared to TPC-B. Even though the index-scans of TPC-C increase the locality of data accesses during the scan operation, in general, the TPC-C transactions access multiple tables with many rows and very little re-use.

Per transaction. Figure 12 shows instruction and data stall cycles per transaction for TPC-C. We observe the sharp decrease in *HyPer*'s *LLC* data stalls compared to per 1000 instructions results, similarly to the micro-benchmark case (Section 4.1.3). *DBMS D*'s instruction stalls are the highest among all the systems showing the large instruction footprint of the traditional disk-based *DBMS D*. *DBMS D* is followed by the disk-based system, *Shore-MT*, and the main-memory optimized OLTP engine *DBMS M*. While *DBMS M*'s instruction stalls are lower than the disk-based systems, they are still considerably large due to its legacy code components.

5.3 Summary

Overall, TPC experiments show similar trends to the micro-benchmark ones. The higher micro-architectural utilization (the higher IPC values) stems from the higher data (mainly TPC-B) and instruction (mainly TPC-C) locality in the TPC workloads compared to the micro-benchmark, which especially the in-memory systems can effectively exploit thanks to their optimized code and data structures.

6. INDEX AND COMPILATION OPTIMIZATIONS, AND DATA TYPES

This section analyzes the impact of index and compilation optimizations the in-memory systems adopt, as well as the impact of the data types, at the micro-architectural level. Among the systems used in this study, *DBMS M* is the only one that allows enabling/disabling the compilation optimizations and using two different index structures; hash index and a variant of cache-conscious B-tree index similar to [17, 18]. Therefore, while we use *DBMS M* for analyzing the impact of index and compilation optimizations, we experiment with all the three in-memory systems (*VoltDB*, *HyPer*, and *DBMS M*) to quantify the effect of different data types.

6.1 Impact of index type and compilation

To quantify the impact of the type of index and compilation on the micro-architectual utilization, we start with the read-only variant of the micro-benchmark and plot the stall cycles per 1000 instructions in Figure 13. The results for the read-write version of the micro-benchmark can be found in Section A.3 of the Appendix. We use the version of the micro-benchmark where we access 10 rows per transaction from the 100GB dataset. As expected, compilation optimizations have significant effect on the instruction stalls, resulting in $\sim 50\%$ reduction regardless of the index type. On the other hand, while the choice of index does not change the instruction stalls behavior significantly, we observe that *LLC* data stalls are $2-4x$ larger for the B-tree index than the hash index. B-trees require traversing the entire index to probe a single row potentially touching many internal nodes of the B-tree. Hash index, on the other hand, directly goes to the hash bucket that corresponds to the probed keys. Therefore, hash index requires fewer random data requests incurring fewer data misses.

We repeat the experiment above using the TPC-C benchmark. Figure 14 shows the stall cycles per 1000 instructions from different levels of the cache hierarchy. Once again, compilation optimizations reduce instruction stalls significantly

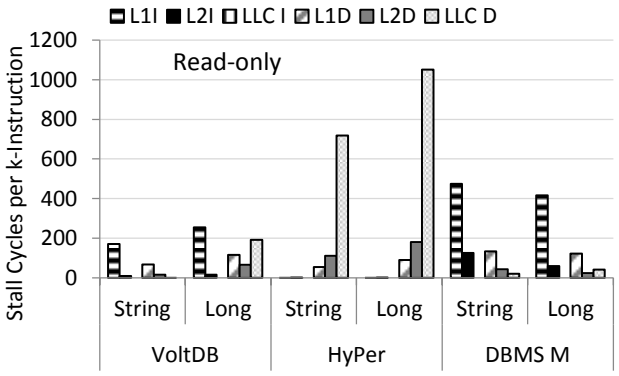

Figure 15: Stall cycles per 1000 instructions for String and Long data types while running the micro-benchmark.

for both index types. However, unlike the results with the micro-benchmark, instruction stalls are much higher for the B-tree index than the hash index without the presence of transaction compilation optimizations. This shows compilation is also effective in eliminating inefficiencies in the instruction stream when using a B-tree index. In the case of data stalls, since the TPC-C benchmark requires fewer random data reads compared to the micro-benchmark, we do not observe significant data stall time in Figure 14 for TPC-C regardless of the index types.

6.2 Impact of data type

To quantify the impact of different data types on micro-architectural utilization, we use the read-only version of the micro-benchmark where we probe 1 row per transaction over a 100GB database. The results for the read-write version of the micro-benchmark can be found in Section A.3 of the Appendix. We modify the micro-benchmark to use two 50 bytes *String* columns instead of two *Long* columns in the table and compare the two versions.

Figure 15 shows the results for *VoltDB*, *HyPer*, and *DBMS M*. We observe that the *LLC* data stalls are lower for *String* data type than for *Long* for *VoltDB* and *HyPer*. This is because larger data items such as 50 bytes *String*s, provide better spatial locality than the smaller, 8 bytes *Long*s. While traversing a B-tree, comparing two 50 bytes of *Strings* would re-use the same cache line more frequently than comparing two 8 bytes of *Long*s. On the other hand, *DBMS M* does not have a significant difference in its data stalls while using *String* or *Long*. This is partly because of the larger instruction footprint of *DBMS M* and partly because of using a hash index structure rather than a B-tree.

6.3 Summary

Overall, we once again corroborate that the transaction compilation optimizations decrease the instruction related stall time fundamentally for OLTP systems by both reducing the instruction footprint and leading to a smoother instruction stream. The choice of index structures mainly affects the data locality especially for workloads that require frequent random data accesses. The data type, on the other hand, can affect the spatial locality of the workload, however, it does not affect the conclusions fundamentally.

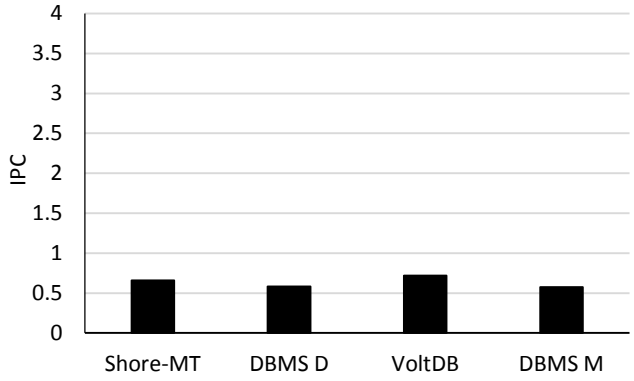

Figure 16: The IPC values for the multi-threaded experiments while running the micro-benchmark.

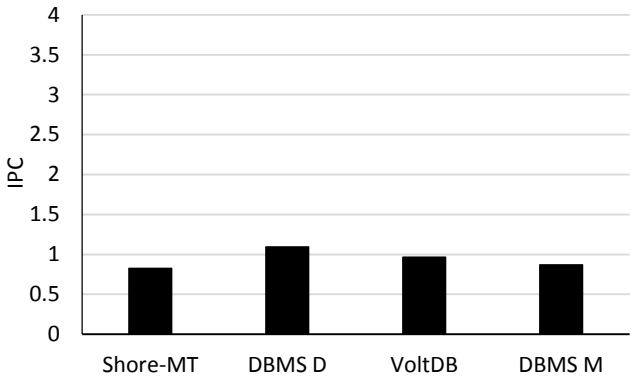

Figure 17: The IPC values for the multi-threaded experiments while running TPC-C.

7. IMPACT OF MULTI-THREADING

This section analyzes the effect of running multiple server side threads on the micro-architectural behavior. The single-threaded experiments aim to present an idealized case for all the systems since it avoids cache invalidations due to data sharing across different worker threads or misleading artificially high IPC values due to threads spinning under possible contention. On the other hand, multi-threaded experiments aim to investigate a more realistic scenario where systems are loaded with multiple threads executing transactions from multiple clients.

Figure 16 and Figure 17 show the IPC values while running the read-only version of the micro-benchmark when reading 1 row, and TPC-C benchmark, respectively. We use a database of size 100GB in both of the experiments. As can be seen from the figures, the IPC values are smaller than one for all the systems under both workloads (except for *DBMS D* running TPC-C) similar to the results from the single-threaded experiments (see Figure 1 and Figure 10).

Figure 18 and Figure 19 show the stall cycles per 1000 instructions while running the read-only version of the micro-benchmark and TPC-C benchmark, respectively. All the systems in Figure 18 and Figure 19 have similar stall cycles results for both the multi-threaded and single-threaded configurations.

As a side note, *VoltDB* adopts further optimizations in its concurrency control mechanism when it knows that there

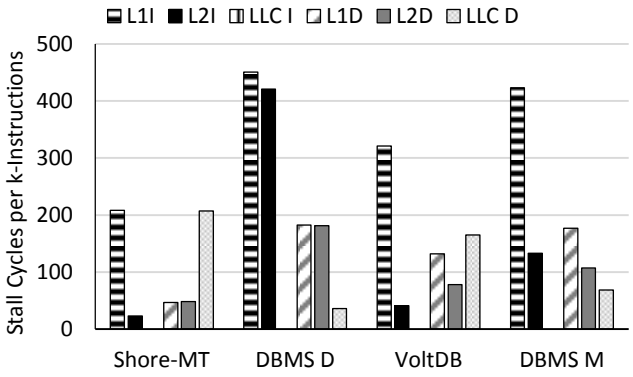

Figure 18: Stall cycles per 1000 instructions for the multi-threaded experiments while running the micro-benchmark.

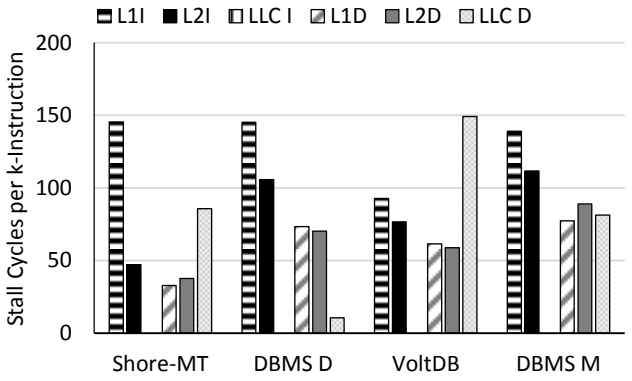

Figure 19: Stall cycles per 1000 instructions for the multi-threaded experiments while running TPC-C.

are no multi-partition transactions (as in the case for all the experiments with *VoltDB* in this paper). If we do not ensure single-site transactions, the instruction stalls of *VoltDB* increase significantly (by $\sim 60\%$).

Summary. Overall, the results in a multi-threaded configuration do not change the high-level conclusions from the results with a single-threaded configuration for the systems under analysis.

8. SUMMARY AND IMPLICATIONS

This section summarizes the highlights of our experimental study and discusses their implications.

In-memory OLTP systems implement a series of optimizations to reduce the instruction footprint and improve cache utilization. However, despite all of the optimizations, they severely under-utilize the micro-architectural features similarly to the traditional disk-based systems. The IPC values barely reach one on a machine that is capable of retiring four instructions per cycle.

Instruction stalls. *DBMS M* incurs the highest number of instruction stalls among the in-memory systems per transaction due to the large amount of legacy code it borrows from the traditional OLTP system it belongs to. On the other hand, *HyPer* is able to reduce the instruction stalls to almost zero thanks to its aggressive transaction compilation. However, high instruction locality amplifies the long latency data stalls resulting in low IPC values.

Data stalls. The data stalls when running OLTP workloads are affected by two factors: (1) whether the index is cache-conscious or not and (2) what the number of random data accesses per unit of time is. We observe that, except for *Shore-MT*, all the systems we analyze adopt some variant of cache-conscious index structures to improve data cache locality. On the other hand, systems that optimize instruction streams by transaction compilation, such as *HyPer* and *DBMS M*, perform more frequent random data accesses. Despite having cache-conscious index structures, they both suffer dramatically from long latency data stalls when requests do not exhibit data locality.

Implications. In this study, we conclude that software-level optimizations do not directly translate into more efficient utilization of micro-architectural resources, and might even hinder it, on modern processors. One needs to optimize the hardware and software together as the next step putting micro-architectural utilization as a high priority goal. This will, in turn, improve not only overall performance but also energy efficiency.

Exhibiting low instruction- and memory- level parallelisms, OLTP workloads are unable to utilize the wide-issue aggressive out-of-order cores that implement complex hardware mechanisms. Majority of the execution cycles go to the memory stalls for bringing either instructions from the *L1* cache or data items from the *LLC*. *L1* instruction cache sizes have been unchanged for the last decade due to the strict latency limitations and we cannot expect them to increase. On the other hand, whatever the size of the *LLC* is, megabytes of *LLC* will not be enough to keep the gigabytes of the data footprint of most standard OLTP benchmarks. Hence, instead of using beefy and complex out-of-order cores consuming large amount of energy, using simpler cores with caching mechanisms tailored toward the instruction and data accesses of typical OLTP applications would lead to higher energy-efficiency with better or similar performance.

9. CONCLUSION

In this paper, we perform a detailed micro-architectural analysis of the in-memory OLTP systems contrasting them to the disk-based OLTP systems. Our study demonstrates that in-memory OLTP system behave very similarly to the disk-based OLTP systems despite all the design differences and lighter storage manager components of the memory-optimized systems. The lighter storage manager components reduce the instruction footprint at the storage manager layer, but the overall instruction footprint of an in-memory OLTP system is still large, which leads to a poor *L1-I* locality and high number of *L1* instruction misses. Even though optimized compilation techniques help in minimizing the *L1* instruction misses, in the absence of the instruction misses the impact of long-latency data misses surfaces resulting in low IPC values.

Acknowledgements

We would like to thank the members of the DIAS laboratory for their constructive feedback and support throughout this work. The work in this paper has been partially funded by the Swiss National Science Foundation, project "200021-146407/1".

10. REFERENCES

[1] TPC transcation processing performance council. http://www.tpc.org/default.asp.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *ISCA*, pages 3–14, 1998.

[4] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control—Theory and Algorithms. *ACM TODS*, 8(4):465–483, 1983.

[5] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012.

[7] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007.

[8] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008.

[9] HyPer. http://hyper-db.de/.

[10] Intel. Intel VTune Amplifier XE Performance Profiler. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[11] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*, pages 15–26, 1998.

[12] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the Hybrid OLTP & OLAP Main-Memory Database System Hyper. *IEEE DEBull*, 36(2):41–47, 2013.

[13] T. Lahiri, M. Neimat, and S. Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE DEBull*, 36(2):6–13, 2013.

[14] P. Larson, M. Zwilling, and K. Farlee. The Hekaton Memory-Optimized OLTP Engine. *IEEE DEBull*, 36(2):34–40, 2013.

[15] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund. High-Performance Transaction Processing in SAP HANA. *IEEE DEBull*, 36(2):28–33, 2013.

[16] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*, pages 38–49, 2013.

[17] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, pages 302–313, 2013.

[18] J. Lindstrom, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE DEBull*, 36(2):14–20, 2013.

[19] MemSQL. http://www.memsql.com/.

[20] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[21] T. Neumann and V. Leis. Compiling Database Queries into Machine Code. *IEEE DEBull*, 37(1):3–11, 2014.

[22] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented Transaction Execution. *PVLDB*, 3(1):928–939, 2010.

[23] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared-memory Systems with Out-of-Order Processors. In *ASPLOS*, pages 307–318, 1998.

[24] Shore-MT. Shore-MT Official Website. http://diaswww.epfl.ch/shore-mt/.

[25] R. Stets, K. Gharachorloo, and L. Barroso. A Detailed Comparison of Two Transaction Processing Workloads. In *WWC*, pages 37–48, 2002.

[26] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of An Architectural Era: (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[27] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE DEBull*, 36(2):21–27, 2013.

[28] P. Tözün, B. Gold, and A. Ailamaki. OLTP in Wonderland – Where Do Cache Misses Come From in Major OLTP Components? In *DaMoN*, pages 8:1–8:6, 2013.

[29] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks – The Obsolete, The Ubiquitous, The Unexplored. In *EDBT*, pages 17–28, 2013.

[30] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, pages 18–32, 2013.

[31] VoltDB. http://www.voltdb.com.

[32] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Streams in Commercial Server Applications. In *IISWC*, pages 99–108, 2008.

[33] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB*, 8(3):209–220, 2014.

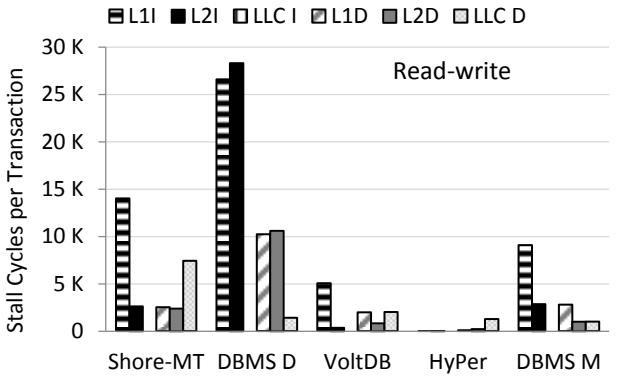

Figure 22: Stall cycles per transaction from the different levels of the memory hierarchy for a database of size 100GB.

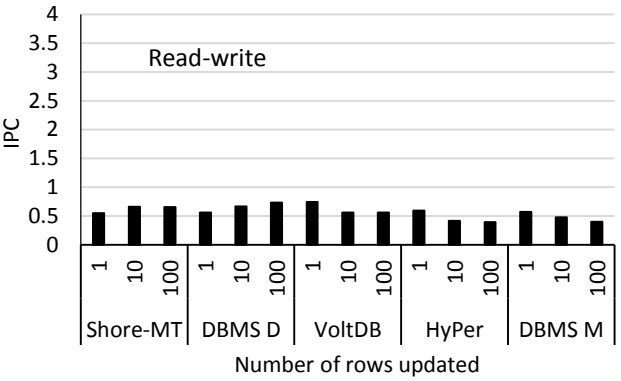

Figure 23: Effect of the amount of work per transaction on the IPC value with a database of size 100GB.

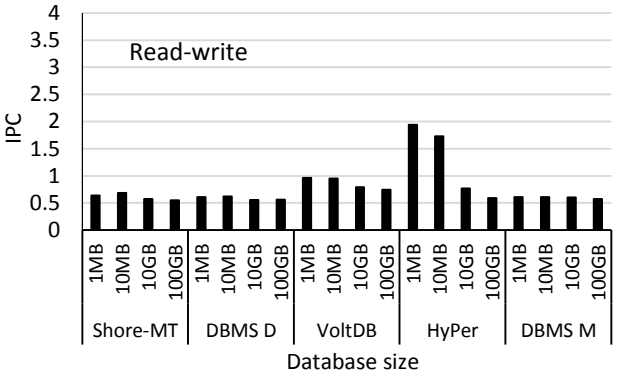

Figure 20: Effect of database size on the IPC value.

APPENDIX

In this appendix, we extend our sensitivity analysis with the results of experiments while running the read-write version of the micro-benchmark.

A. READ-WRITE MICRO-BENCHMARK

A.1 Sensitivity to Data Size

We start with the experiment where we increase the data size from 1MB, to 10MB, 10GB and 100GB. Figure 20 shows the IPC values when we run the read-write version of the micro-benchmark for 1 row.

The trends in IPC value do not differ much across read-only and read-write benchmarks, although, in general, the IPC values are higher for the read-only benchmark (see Section 4.1.1). The instruction footprint of the read-write micro-benchmark is larger than that of the read-only benchmark since, in the read-write benchmark, transactions both read and update the probed row as opposed to the read-only benchmark only reading the probed row. Therefore, all the systems, except for *HyPer*, have poorer instruction locality while running the read-write micro-benchmark (Section 4.1.2). This leads to the lower IPC value for this benchmark. *HyPer*, on the other hand, suffers less from the instruction stalls and more from the data stalls (Section 4.1.2), so it exhibits a higher IPC value for the read-write micro-benchmark compared to the read-only one.

As is for the read-only version of the micro-benchmark, IPC values are similar to each other when the data size is less than the capacity of the last-level cache (LLC), i.e., 20MB. As the data size increases to 10GB and 100GB exceeding the LLC capacity, IPC values decrease as the misses from *LLC* require accessing the main memory. This effect is dramatic for *HyPer* due to its sensitivity to the data stalls, and less observable for *DBMS D* and *DBMS M* due to their large instruction footprints. Even for small data sizes, e.g., 1MB and 10MB, the large instruction footprint of *DBMS D* and *DBMS M* causes high instruction stalls resulting in low IPC.

Figure 21 shows the stall cycles per 1000 (k-)instructions as we increase the dataset size for the read-write micro-benchmark. We observe that the instruction stalls are higher for the read-write micro-benchmark compared to the read-only micro-benchmark showing the larger instruction footprint of the read-write micro-benchmark. Similar to the read-only micro-benchmark, instruction stalls dominate the overall stall time.

Lastly, Figure 22 shows the stall cycles per transaction. Once again, we observe that the instruction stalls are higher for the read-write micro-benchmark compared to the read-only micro-benchmark.

A.2 Sensitivity to Work per Transaction

This section presents the results while running the read-write version of the micro-benchmark as we increase the amount of work per transaction by increasing the number of rows updated per transaction from 1, to 10 and 100.

Figure 23 plots the IPC values as the amount of work per transaction increases on the x-axis. We observe similar trends to the read-only micro-benchmark where IPC values increase for the disk-based systems (*Shore-MT* and *DBMS D*), and decrease for the in-memory systems (*VoltDB*, *HyPer*, and *DBMS M*). Moreover, IPC values are in general lower for the read-write micro-benchmark compared to the read-only micro-benchmark.

Figure 24 plots the stall cycles per 1000 (k-)instructions. We observe that the instruction stalls are higher and the data stalls are lower for the read-write micro-benchmark compared to the read-only micro-benchmark. Similarly to the read-only case, instruction stalls decrease as the number of rows updated per transaction increases due to higher locality within the instruction stream. *DBMS D*'s instruction stalls are high even when probing 100 rows implying the large instruction footprint of the commercial disk-based

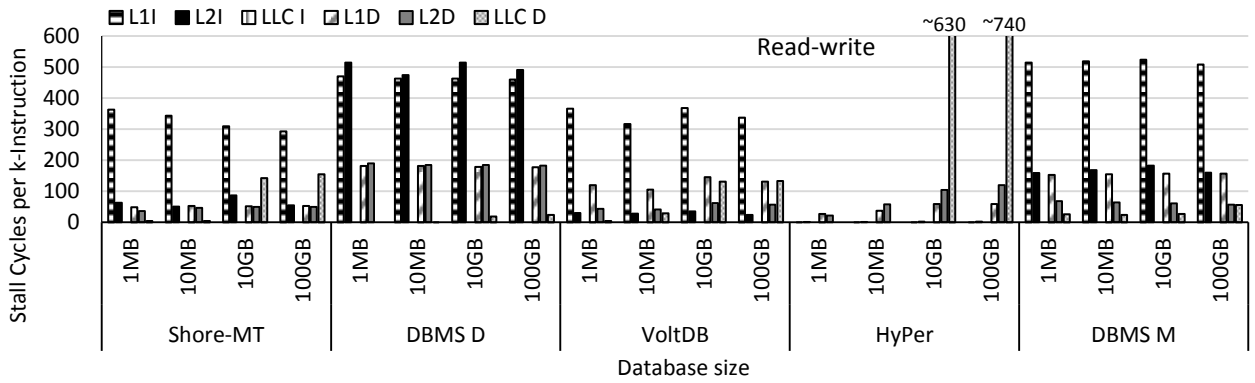

Figure 21: Stall cycles per 1000 instructions from the different levels of the memory hierarchy as we increase the database size.

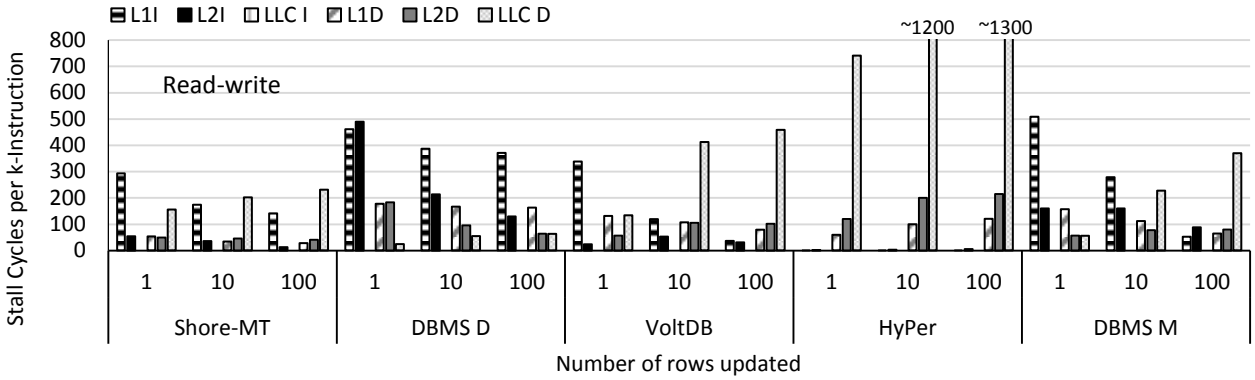

Figure 24: Stall cycles per 1000 instructions from the different levels of the memory hierarchy as we increase the amount of work done per transaction with a database of size 100GB.

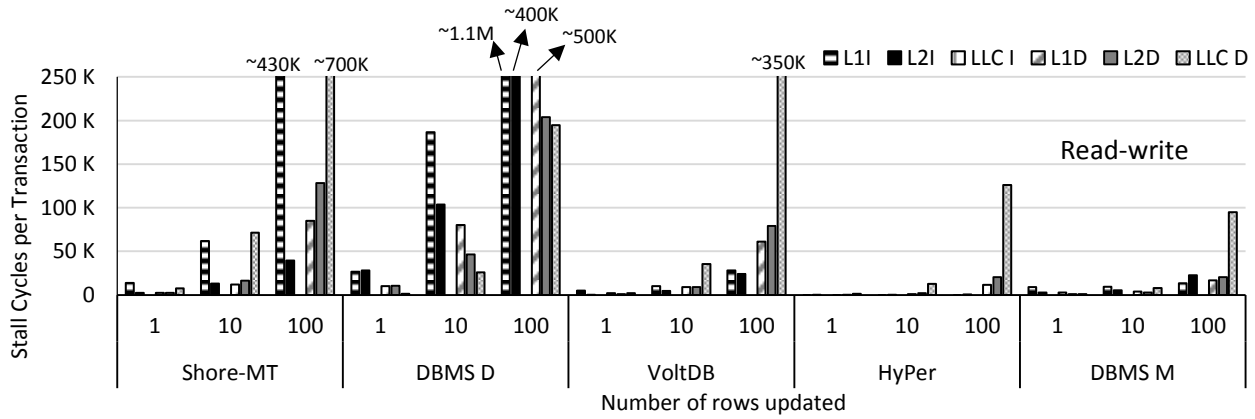

Figure 25: Stall cycles per transaction from the different levels of the memory hierarchy as we increase the amount of work done per transaction with a database of size 100GB.

OLTP system. *DBMS M*'s instruction stalls dominate for probing 1 and 10 rows showing the high overhead of the legacy code *DBMS M* uses from the traditional disk-based OLTP system it belongs to. For probing 100 rows, however, its instruction stalls become significantly smaller.

Lastly, Figure 25 shows the stall cycles per transaction as we increase the number of rows updated per transaction on the x-axis. Once again, we observe higher instruction stalls for the read-write micro-benchmark compared to the read-only micro-benchmark. Moreover, as the number of rows updated per transaction increases, the instruction stalls significantly increase for *Shore-MT* and *DBMS D*, and stay the same or slightly increase for *VoltDB*, *HyPer*, and *DBMS M*. The data stalls, on the other hand, increase almost linearly for all the systems as the number of rows updated increases. We observe that the data stalls of Shore-MT are 2-3.5x higher than the other systems suggesting that all the

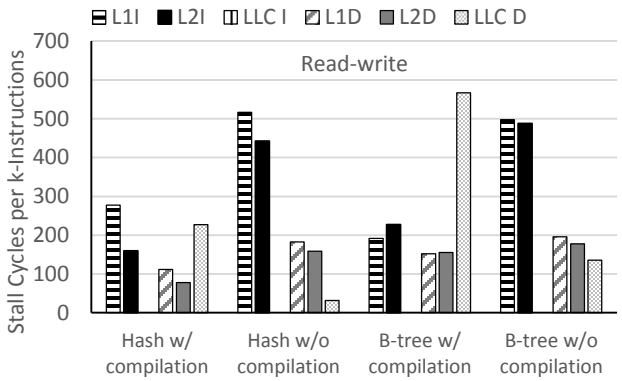

Figure 26: Stall cycles per 1000 instructions for different index structures with and without compilation optimizations when running the micro-benchmark.

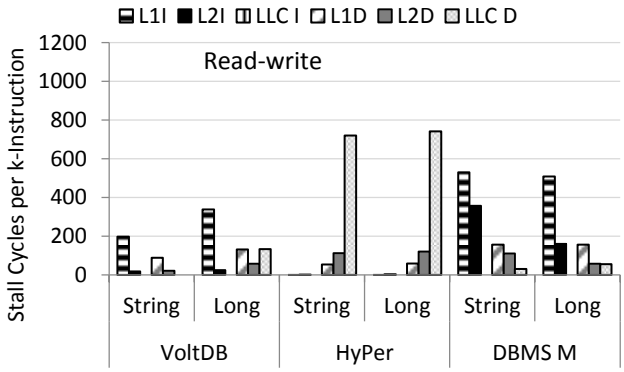

Figure 27: Stall cycles per 1000 instructions for String and Long data types when running the micro-benchmark.

other systems implement some variant of cache-conscious index structure.

A.3 Index, Compilation, and Data Type

Finally, this section presents the results while running the read-write version of the micro-benchmark as we use different index and compilation optimizations as well as different data types.

Figure 26 depicts the stall cycles per 1000 (k-)instructions for different index and compilation configurations for the read-write micro-benchmark. We observe similar trends to the read-only micro-benchmark. The instruction stalls decrease significantly when we use the compilation optimization for both hash and B-tree indexes, while different index structures do not affect the instruction stalls. On the other hand, B-tree index causes significantly higher data stalls than the hash index.

Figure 27 plots the stall cycles per 1000 (k-)instructions while running the read-write micro-benchmark with different data types, i.e, *String* and *Long*. We observe the slightly higher data stalls for *Long* than for *String* when using *VoltDB* and *HyPer*. Although this effect is similar to the one we observe while running the read-only micro-benchmark, the differences between the data stalls for *String* and *Long* are lower for the read-write micro-benchmark. This is because the read-write micro-benchmark has better spatial locality than the read-only micro-benchmark since the read-write micro-benchmark, after reading the cache line corresponding to the probed value, accesses the same cache line to write the updated value. *DBMS M*, on the other hand, exhibits similar data stalls for both of the data types, and has higher *L2* instruction stalls for the *String* data type.